Artist Friendly Fracture Modelling

TNXD20 - Master's Thesis

Erik Johansson-Evegård

Sunday 22nd January, 2012

Abstract

Destruction is one of the key aspects of visual effects. This report describes the work that was done to create a production ready pre-fracture modelling plug-in for Maya. It provides information on what methods that can be used to create a robust plug-in and various techniques for sampling points to create interesting fracture patterns using the Voronoi diagram. It also discusses how this work can be further built on to create an even better plug-in.

Contents

1. Introduction **1**
- 1.1. Visual Effects . . . 1
- 1.2. Fracturing . . . 1
- 1.3. Motivation . . . 2
- 1.4. Aim . . . 2
- 1.5. Outline Of Report . . . 2

2. Background **3**
- 2.1. Pre-Simulation Fracturing . . . 3
- 2.2. Simulation Based Fracturing . . . 3
- 2.3. Polygon Clipping . . . 3
- 2.4. Closing A Clipped Mesh . . . 4

3. Theoretical Prerequisites **5**
- 3.1. Voronoi Diagram . . . 5
- 3.2. Polygonal Mesh . . . 6
- 3.3. Mesh Clipping . . . 6
- 3.4. Mesh Capping . . . 10
- 3.5. Object properties . . . 11

4. Implementation **13**
- 4.1. Algorithm Implementation . . . 13
- 4.2. Data Structures . . . 13
- 4.3. Autodesk Maya Implementation . . . 14
  - 4.3.1. Interface and Usage . . . 14
  - 4.3.2. Fracture Points Sampling Techniques . . . 15
    - 4.3.2.1. Random . . . 16
    - 4.3.2.2. NURBS Sphere . . . 16
    - 4.3.2.3. Particles . . . 16
    - 4.3.2.4. NURBS Curve . . . 16
    - 4.3.2.5. Radial Sampling . . . 17
  - 4.3.3. Clustering . . . 17

5. Result **18**
- 5.1. Random . . . 18

5.2. NURBS Sphere . . . 19
5.3. Particles and Crack-Maps . . . 19
5.4. NURBS Curve . . . 20
5.5. Radial Shatter . . . 20
5.6. Clustering and Visualization . . . 21
5.7. Performance . . . 22
5.8. Simulation Fracturing Comparison . . . 23
5.9. In Use . . . 24

6. Discussion **25**

7. Conclusion **26**
7.1. Further Work . . . 26
7.1.1. Threading . . . 26
7.1.2. Procedural fracture . . . 26
7.1.3. Arbitrary Cut Planes . . . 27
7.1.4. Command Implementation . . . 27

A. User Manual **31**
A.1. Parameters . . . 31
A.2. Usage . . . 31
A.3. Random and Radial Mode . . . 32
A.4. NURBS Sphere, NURBS Curve and Particle Mode. . . . 32

1. Introduction

This report presents the Master Thesis work performed at Important Looking Pirates VFX in Stockholm between April and November in 2010. The thesis is needed to fulfil a degree in Master of Science in Media Technology and Engineering from Linköping University, Sweden. The introduction chapter will present the motivation, aim for the thesis and outline the report and give a brief introduction to visual effects and fracturing.

1.1. Visual Effects

Visual Effects (VFX) is the process of adding content to or modifying existing film footage to create visuals that would have been costly or impossible to capture with a regular camera. This often involves the destruction of buildings and fracture of smaller objects.

1.2. Fracturing

Fracture and destruction is a key part of today's VFX productions. There are currently two branches of work-flow commonly used; pre-fracturing and simulation based fracturing. Simulation based fracturing have previously foremost been used in building construction to calculate tension in buildings but have as computers becomes faster started to become a viable solution for visual effects. It does still however have several drawbacks where speed and lack of artist influence are most noticeable. In simulation based fracturing the fracturing is performed as the simulation progresses. When pre-fracturing the individual pieces of the object are created before simulating the actual destruction which gives artists possibilities to create specific looks and enables the use of fast methods for performing rigid body simulations.

The work-flow of creating a destruction effect using pre-fracturing methods can be described as a set of steps:

- Step 1. Create the original object using normal modelling and shading techniques.
- Step 2. Fracture the object into a set of small objects using a chosen method as manually cutting it up, Voronoi fracturing or other technique.
- Step 3. Rig the effect by setting up rules for activation of pieces and constraints.
- Step 4. Simulate the destruction using a rigid body simulator.

- Step 5. Optionally add additional dust and debris using information gathered during simulation.

1.3. Motivation

Autodesk Maya the currently leading product for visual effects and the main product in the ILP pipeline lacks good features that enables artists to quickly get the desired look and result when fracturing models. Therefore it is of need to develop a faster work-flow that focuses on speed and ease of use and can be used in production where results are expected to be delivered quickly.

1.4. Aim

This thesis will aim at developing a method for pre-fracturing a object and implementing it as a production ready plug-in for Autodesk Maya using C++. It needs to be capable of handling the different polygonal models that are provided to the artist. The models might contain polygons that are both convex, concave and contain holes. Surface information as shaders, normals and UV-coordinates also need to be handled and transferred to the fractured object. Turnovers between iterations should be quick so that the desired result can quickly be achieved.

1.5. Outline Of Report

In Chapter 2 currently used production techniques are explained, their pros and cons are discussed and from this a choice of technique have been made. Chapter 3 will describe the method and theories used in the implementation. Implementation details will be given in 4 and results presented in Chapter 5. The report ends with a discussion of the results in Chapter 6. Chapter 7 concludes the report by presenting further work.

2. Background

This thesis is focused on pre-simulation fracture modelling 2.1 but there is also simulation based fracturing 2.2 where real life material properties are taken into consideration. In the background chapter information on two important parts of pre-fracturing that can raise problems will also be discussed. These problems are polygon clipping 2.3 when clipping the mesh and closing a clipped mesh 2.4.

2.1. Pre-Simulation Fracturing

The most used and established technique for creating destruction effects in today's visual effects is for artists to model the individual pieces before simulating them as individual rigid body objects or using constrains to tie the pieces together until a strong enough force breaks them apart [9], [19]. The modelling of the pieces can be done manually or using mathematical methods. Among the mathematical methods the Voronoi diagram is the most widely used because of it's capabilities of mimicking the look of a fractured brittle object as stone, glass or concrete.

2.2. Simulation Based Fracturing

Simulation Based Fracturing using FEA (Finite Element Analysis) method of calculating stress in materials using tetrahedrons is starting to become an alternative for previously used methods in VFX. Been previously only used in construction it becomes a viable alternative as the computers are becoming faster. Currently the only commercial solution is DMM (Digital Molecular Matter) developed by Pixelux [1]. DMM implements their own optimized version of J.F. O'Briens research on graphical modelling of ductile and brittle materials [20], [21]. While a realistic fracturing through simulation can yield very realistic results of breaking complex materials it is very hard to direct and control if a specific result is sought after as look and movement of hero pieces.

2.3. Polygon Clipping

Upon clipping of a polygon most algorithms first consider what vertices are in the correct half-space and from this information clip edges if they are intersecting the clip-line or clip-plane. Where the algorithms differentiate is how they handle closing the polygons where they have been clipped. One common and easy to grasp algorithm is

Sutherland-Hodgman clipping algorithm [23]. They close polygons by keeping a CW (Clock Wise) or CCW (Counter Clock Wise) ordered polygon. If the order contains two intersection-points following each other an edge is inserted between them. This works well for convex polygons but can create overlapping edges and zero area parts if the polygon is concave. It also have no way of dealing with polygons that contains holes. This makes it not usable as it will create render artefacts and might generate problems during simulations. Weiler-Arthon clipping algorithm [10] can handle any shape of polygons by keeping information about if the intersections are leaving or entering the polygon in a list and also storing the same information for the polygon used for clipping in a separate list. It then creates the resulting polygons by traversing the polygon and if an intersection point is found it switches to the other lists corresponding intersection point and continues the traversal. In Graphic Gems V [13] a simplified version of the Weiler-Arthon algorithm is presented where a separate list for the clipping polygon is not necessary but instead misses the functionality of clipping polygons with holes. Extending the algorithm from Graphic Gems V with theory from Weiler-Arthon a algorithm that can handle both concave and polygons with holes can be constructed.

2.4. Closing A Clipped Mesh

The closing of a clipped mesh is no common problem compared to the actual clipping of the mesh which have received much research in the purpose of removing parts that are not relevant when rendering. David Eberly Clipping a Mesh Against a Plane[8] suggests a trivial approach of using the information gathered while clipping the mesh which stores each edge that have been used to close a clipped polygon. It holds that these should be the edges constructing the resulting polygon need to cap the mesh. The information about how they connect is however missing. Eberly's approach is to simply sort the edges so that a starting- and end-point are put together. This works for meshes that are convex but breaks if the mesh is concave and that more than one polygon is necessary to successfully close the mesh. For solving this theory have been borrowed from the graph field. If considering all edges to be parts of a directed graph where the connectivity information is missing Donald B. Johnson's paper on finding elementary circuits can be applied. The output of elementary circuits are the polygons and holes needed to successfully close a mesh being both concave and with holes. It is of importance to have an algorithm that can handle all cases as a concave mesh can be clipped to create a mesh containing a hole.

3. Theoretical Prerequisites

This chapter will describe algorithms and techniques used in the implementation. A brief overview of used surface representation and technique for fracturing a model into small pieces is presented. It also describes how different research have been combined to handle problems with concave polygons and holes in polygons when clipping or capping. Clipping a mesh is a key part of the thesis as each piece of the fractured object is generated by clipping parts of the model with clip planes positioned between site points until only the area belonging to the currently generated pieces site point is left. The capping of the mesh is done after each time the mesh is clipped to create a new surface where there is none. This is because of the explicit representation of the model having no information about the inner workings of the model and only carries information about the outer visible surface.

3.1. Voronoi Diagram

For modelling the new pieces from the original object the Voronoi diagram is used. The definition is taken from [15] and was introduced in 1908 by Georgy Voronoi [16]. A set of points $P = p_1, p_2, ..., p_n$ in $\mathbb{E}^2$-space are called sites and can be used to partition a $2D$ plane where each region in the plane is assigned to their nearest site point according to:

$$V(p_i) = \{x : d(p_i, x) \leq d(p_j, x), \forall j \neq i)\}. \tag{3.1}$$

$d(x, y)$ is the distance between point x and point y. This definition also holds for $\mathbb{E}^3$-space.

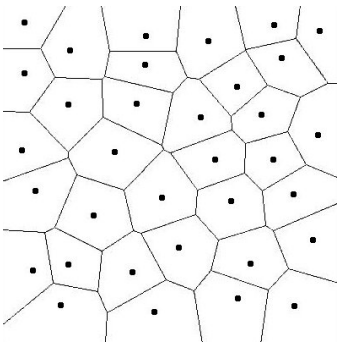


Figure 3.1.: $2D$ Voronoi Diagram

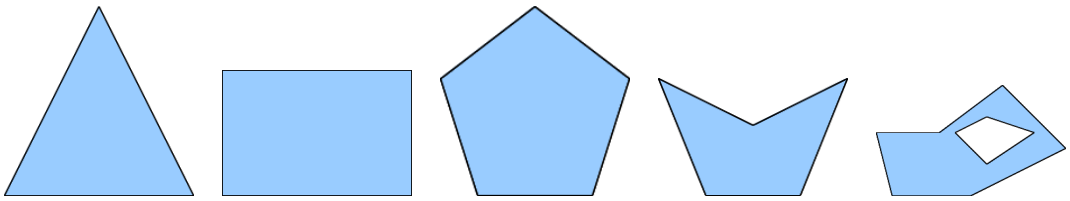
For computing the Voronoi diagram an algorithm calculating intersections with half-spaces is used. In $\mathbb{E}^2$ an half-space is a half-plane defined as all points on one side of an infinite straight line. In $\mathbb{E}^3$ the line becomes instead an infinite plane but the same theory can be applied.

By repetitive intersection calculations against $n-1$, where n is the amount of site points, clipping planes each region is calculated separately. This is further explained in section 5.2.1 and 5.4.1 of [15].

Using the Voronoi diagram for fracture modelling is a proven and widespread technique [12].

3.2. Polygonal Mesh

3D models in VFX are in most cases explicit surfaces consisting of polygons. Explicit surfaces can also be represented using parametric functions as NURBS (Non-Uniform Rational B-Spline), bump maps or displacement maps. A collection of polygons is called a mesh and can be represented in multiple ways. One of the most common and also used in Autodesk Maya API (Application Programming Interface) [3] is the Face-Vertex representation. In a Face-Vertex representation all vertices are stored in a list as $V = \{v_o, v_1, v_2, ..., v_n\}$ and each face stores the indices it's vertices.



(a) Triangle (b) Quad (c) Convex (d) Concave (e) Polygon with a hole

Figure 3.2.: Polygon types

Ideally when performing clipping on a polygon mesh it only consists of convex polygons as in figure 3.2(c) for which there are trivial and easy to implement algorithms available. However even if an initial step of pre-processing the mesh to only consist of convex polygon both concave 3.2(d) and polygons with holes 3.2(e) can be introduced during the mesh clipping. Therefore methods considered for clipping the mesh needs to able to handle both concave polygons as well as holes. How the holes are handled together with the Face-Vertex mesh representation is further explained in section 4.2.

3.3. Mesh Clipping

Using the algorithm for clipping concave polygons from [13] and extending it to handle polygons with holes using [10] gives an algorithm that is able to handle most models that

are provided by artists. A limitation is that the model is required to be manifold for the capping algorithm described in section 3.4 to work properly.

The algorithm from [13] takes advantage off that in practical use when clipping a polygon it is observed that only one of the resulting polygons is of interest and the other can be disregarded. Requirements for the algorithm is that the vertices of the polygon is represented in either CW or CCW order. Holes in polygons are represented by using a reverse order of the vertices as in [10].

First all edges of the polygon are checked for intersections against the plane by calculating the distances for the edges vertices to the clip-plane. Edges having both vertices in the outside half-space are disregarded and edges with both vertices inside are kept. For edges intersecting the plane the intersection point between the edge and plane is calculated and inserted into a list. A polygon can contain multiple holes. For the holes if none of the edges of the hole is intersected by the clip-plane and are in the inside half-space the hole is considered being part of the new polygon and if all the edges are in the outside half-space the hole is disregarded. If the edges of hole intersect with the clip-plane the edges will be part of the new polygon and intersection points are added to the same list of intersection points.

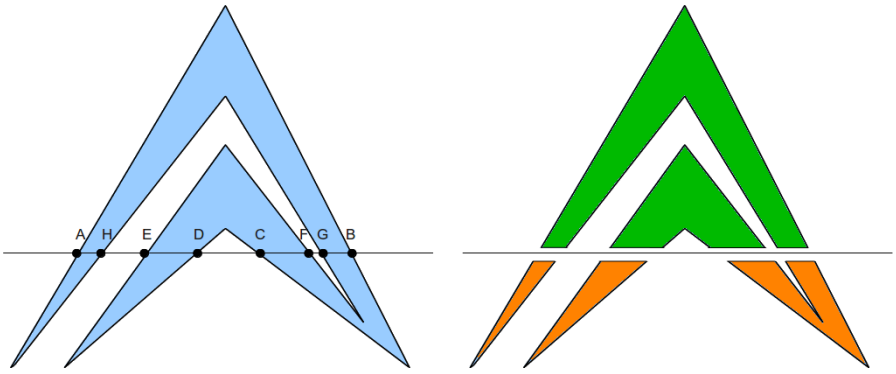


Figure 3.3.: Reentrant clipping of a concave polygon. Figure from [13]

Through the Jordan Curve theorem it can be derived that all intersection points of a polygon come in pairs. From figure 3.3 it can be observed that intersection points C and F form a pair/edge but from the list of intersection points this is not interpretable in it's current state but it is known that the list holds both C and F.

To retrieve the pairs/edges that will be needed to close the polygon(s) after it has been clipped choose two random points, X and Y, from the list containing the intersection points. Using X and Y we create a straight line on which all intersection points are positioned. Choosing X as the origin of the line the signed distance to X for all intersection points is calculated and the list is sorted using the distance. After the intersection points have been sorted it will contain the pairs/edges necessary to close the clipped polygon. For the polygon in figure 3.3 this is (A, H), (E, D), (C, F), (G, B) and we call them links.

For avoiding confusion on how to treat vertices residing exactly in the plane the algorithm borrows theory from [17] which treats vertices in the plane as being inside.

The final step is to close the gaps in the clipped polygon using the links. Each vertex points to the next. By starting at some vertex and stepping around the polygon until returning to the start vertex the new polygon is created. When stepping on an intersection point a lookup is made to find the matching link and the stepping continues from there. As holes were initially reversely ordered compared to regular polygons their order becomes correct for the purpose after being clipped.

The full method used is outlined as pseudo code in algorithm 3.3.1.

Algorithm 3.3.1 Clipping a Polygon Mesh against a Plane

V:Array of vertices v
E:Array of edges e consisting of v_0 and v_1
F:Array of polygons f
P:Implicit representation of plane

for all vertices v in V **do**
  Compute signed distance d between v and P.
end for

for all edges e in E **do**
  if $d(v_0) \leq 0$ **and** $d(v_1) \leq 0$ **then**
    Remove e from the polygon it belongs to.
    Mark polygon f containing e clipped.
    If no edges left in f remove it.
    Edge is completely in the plane or the incorrect half-space.
    continue:
  else if $d(v_0) \geq 0$ **and** $d(v_1) \geq 0$ **then**
    Do nothing.
    Edge is in the correct half-space.
    continue:
  else
    e intersects the plane.
    Calculate intersection point i.
    if $d(v_0) < 0$ **then**
      $v_0 = i$
    else
      $v_1 = i$
    end if
    Mark polygon containing e clipped.
  end if
end for

for all polygons f in F **do**
  if f have been clipped **then**
    Perform algorithm 3.3.2
  end if
end for

Algorithm 3.3.2 Close polygon that might be concave and contain holes.

$V1$: Array of vertices v for current polygon and vertices from holes in the polygon that have been clipped. Each v got a `v->next` pointing to following vertex in the original ordered structure.
$V2$: Array of vertices v creating the closed polygon.
F: Array of $V2$.
L: Map of links l that will be used.

```
while V1 not empty do
  Choose some vertex in V1 to be starting vertex v->start.
  v = v->start
  while v->start != v->next do
    Add v to V2 and remove from V1.
    if v->next in L then
      Add v->next to V2 and remove from V1.
      Set v to corresponding link vertex.
    else
      v = v->next
    end if
  end while
  Add V2 to F and empty V2.
end while
```

3.4. Mesh Capping

Using the links for each polygon from section 3.3 and treating them as a directed graph as in figure 3.4 the algorithm presented by Donald B. Johnsson [11] can be used to create the polygons necessary to cap the mesh where the clip-plane has cut it. One drawback is that the resulting polygons can be both actual polygons and also holes inside of an polygon. To detect if a polygon is a hole it's normal is compared to the cut-planes. If it points in the opposite direction it is considered a hole. The surface normal is calculated using Newell's method [14]. Then the winding number [7] is used to determine which polygon that contains the hole by choosing any of the holes vertices and testing if it is inside an polygon.

An efficient search algorithm to find the elementary circuits of a graph [11] is an algorithm for finding elementary circuits in a directed graph. The definition for elementary paths and elementary circuits taken from [18] is as follows:

> An elementary path contains each vertex at most once in its specification. An elementary circuit is an elementary path with the exception that the first and last vertices of the path are the same.

By finding the elementary circuits of the directed graph we also find the elementary paths which contain the vertices that will become the new polygons. In figure 3.4 an elementary circuit is outlined in red. By using this algorithm it is guaranteed that no self intersecting polygons are created and it is possible to find multiple polygons which shares vertices.

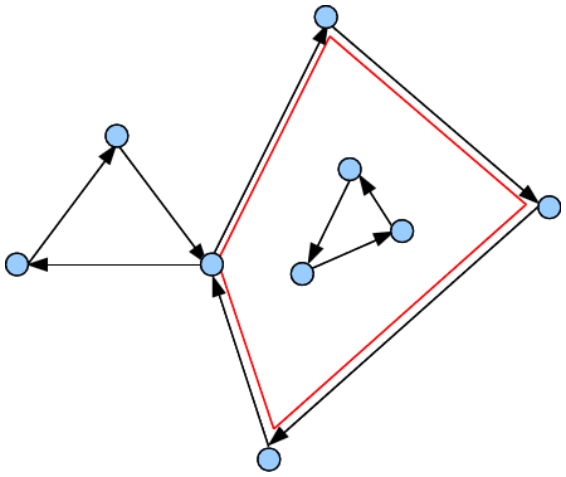


Figure 3.4.: Directed Graph

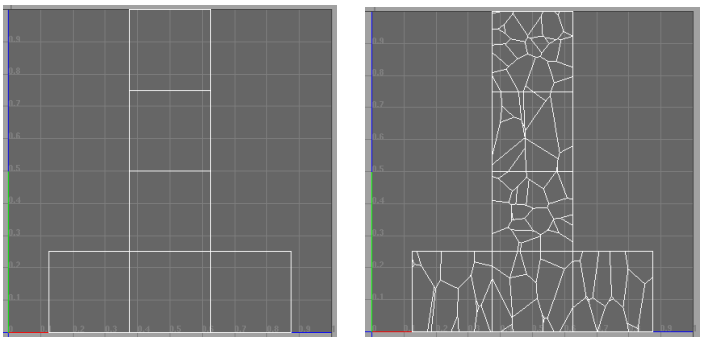
A stack of vertices built from a root vertex s is used to find the elementary circuits. When a vertex is added to the stack it is considered blocked. If the algorithm returns to a blocked vertex v before returning back to s the vertices that have been added on top of the stack after v are removed and the algorithm continues down another path. In [11] pseudo-code for the algorithm can be found.

Calculating the winding number is a 2D operation therefore the polygon to be tested is projected onto a plane positioned either on the X,Y or Z axis that yields the largest area possible for the polygon. This is determined by examining the normal of the polygon and then ignoring the axis with the highest absolute value. It is simple, fast and works for this type of application. After the polygon has been projected a ray is sent horizontally from the point that is going to be tested. What axis to be considered horizontally can differ depending on the projection but for the general case of looking at the x and y components on the point x is considered horizontal. If the ray intersects an edge where the endpoint is above the ray 1 is added to the winding number and if the endpoint is below 1 is subtracted. For all cases where the winding number is different from zero the point is inside of the polygon.

3.5. Object properties

The original object contains a set of surface properties as normals, UVs and shaders. Clipped polygons keep their shader information and new polygons introduced during the mesh capping are assigned a new shader.

Normals and UVs for intersection points are calculated by linear interpolation along the edge where the point have been introduced.



(a) Original UV coordinates (b) New UV coordinates

Figure 3.5.: UV coordinate interpolation

4. Implementation

4.1. Algorithm Implementation

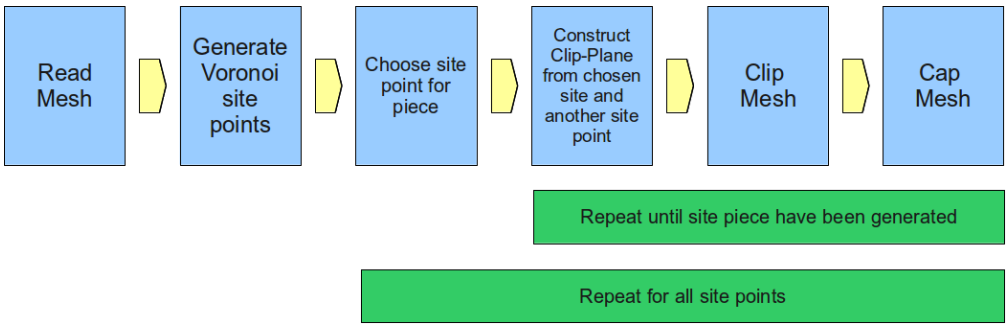

Read Mesh
Generate Voronoi site points
Choose site point for piece
Construct Clip-Plane from chosen site and another site point
Clip Mesh
Cap Mesh
Repeat until site piece have been generated
Repeat for all site points

Figure 4.1.: Flow chart for fracture algorithm.

Figure 4.1 outlines the basic algorithm for the fracture generation. In the first step the mesh is fed into the local data structure. From information about the mesh, settings and objects selected in the scene Voronoi site points are generated. For each site point the same operations are repeated until a fractured piece have been generated. Using the chosen site point a clip-plane is constructed half-way between the current site and another site point. With this clip-plane the mesh is clipped using the theory from section 3.3 and then capped to fill the created hole using theory from section 3.4. The clip-plane generation, mesh clipping and capping is repeated for all site points with exception of the site which is currently being generated.

4.2. Data Structures

Data structures in the implementation takes heavy use of the C++ STL and OpenGL Mathematics (glm) [4] for efficient storage and high performance. Vertices, edges and polygons are stored using std::vector.

The basic mesh data structure is as: A vertex is represented by a glm::vec3 for position. Edges stores the indices for it's vertices as length two integer array. Polygons keep a integer std::list of it's edges indices and an integer std::vector of the indices of the holes it contains.

For both vertices, edges and polygons the data structures holds a boolean representing whether or not they are part of the current piece being generated.

As normals and and UV coordinates can be edge specific, meaning a vertex can have different UV coordinates for different edges, they are stored for each edge.

Further the data structure holds various integers representing indices used for closing polygons when traversing the polygon and as well for the elementary circuit algorithm used during the mesh capping part of the implementation.

4.3. Autodesk Maya Implementation

The thesis have been implemented using C++ for mesh operations and MEL for the user interface. Within the Maya API there is also the possibility of using python but the performance is noticeably lower.

Included in the plug-in is two Maya commands (voro, voroDone) and one Maya node (voroNode). voro creates the voroNode and performs the necessary connections in the DG-Graph, voroNode performs the fracturing of the object and voroDone is used to bake the final pieces and delete unnecessary nodes from the DG-Graph.

For interface generation a MEL node template script is used to generate sliders, buttons and dividers to create figure 4.2. Inside of Maya the attribute spreadsheet can also be used as a secondary interface for the nodes allowing batch updating and changes of settings for multiple objects simultaneously.

4.3.1. Interface and Usage

The interface has been implemented to be as clutter free as possible and activating various features is done by purely selecting them in the view-port. To avoid automatic updates when a setting is changed special rules has been set-up for the node to make it possible for manual update. All settings are also possible to modify through the variable spreadsheet in Maya and update of all nodes at once without pressing update button on each individual is possible. More information on usage is available in appendix A.

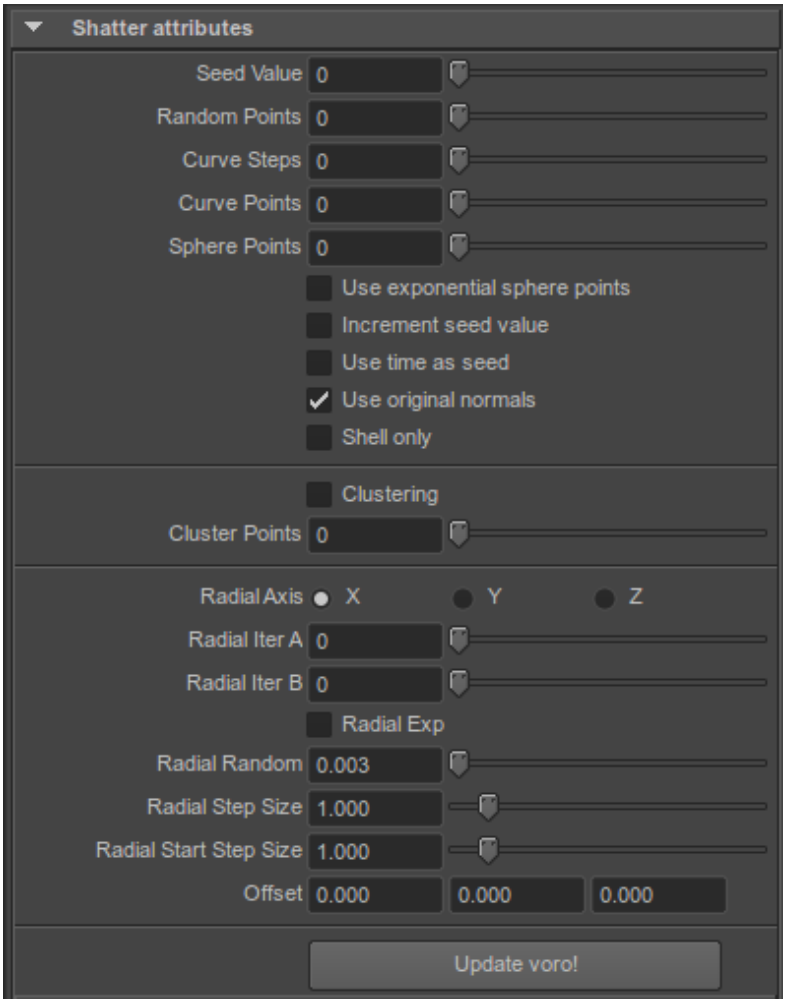


Figure 4.2.: Node interface

Seeding for random generation for different sampling techniques 4.3.2 can be done in two different ways. Either by using an integer as seed to be able to recreate a specific result or by using current time to always get new random values.

For easy distinguishing one piece from another if enabling polygon colouring each piece is visualized using a random colour as in figure 5.6.

4.3.2. Fracture Points Sampling Techniques

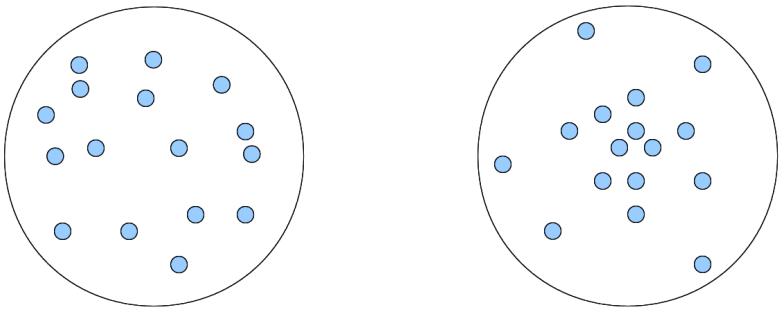
To achieve various looks for the fractured pieces the Voronoi site points have been implemented to be sampled from the scene in different ways.

4.3.2.1. Random

The default way of sampling site points is random sampling within the bounding box of the object to be fractured. When using the bounding box no consideration needs to be taken to that the site points are in object space of the soon to be fractured object. Random sampling creates the classic almost uniform Voronoi look. Figure 5.1 shows the result.

4.3.2.2. NURBS Sphere

By positioning of NURBS spheres the distribution of site points in specific regions can be controlled. Using the NURBS spheres implicit definition points are randomly sampled within it's volume as in figure 4.3(a). The points are transformed to the objects local space using transformation information from the NURBS sphere and the object. Points can also be sampled exponentially as in figure 4.3(b). This creates a higher density of points towards the centre of the sphere. A comparison of pieces generated with exponential versus default is shown in figure 5.2.



(a) Default NURBS sphere sampling (b) Exponential NURBS sphere sampling

Figure 4.3.: NURBS sphere sampling of points

4.3.2.3. Particles

A common work-flow in other applications is using a particle emitter and then the particles as site points. This is also supported and is activated by simply selecting particles and the voroNode simultaneously. From this it is easy to use crack-maps on an object by using a texture to specify where to emit particles as in figure 5.3.

4.3.2.4. NURBS Curve

In a similar fashion as section 4.3.2.2 NURBS curves can be used. Points are sampled in steps a long the curve as in figure 4.4. The distance away from a the curve a point can be sampled is dependent on the step size. The step size can be considered the radius from the step point of which random points are generated. By drawing and aligning a curve

in the object paths of fracture can be created as in figure 5.4. By varying the length of the steps along the curve the width of the fractured path can be controlled.

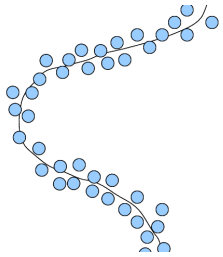


Figure 4.4.: NURBS Curve sampling

4.3.2.5. Radial Sampling

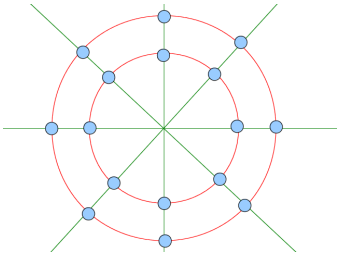


Figure 4.5.: Radial sampling

For radial sampling polar coordinates are used to sample points in steps, shown as green, around a set of circles, shown as red in the figure 4.5, with various radius. By adding random noise of a chosen amplitude the pattering in figure 5.5 can be created.

4.3.3. Clustering

Pieces can also be clustered together to form more irregularly shaped pieces. This is done by sampling a second set of points and assigning each piece that have been created to the closest point. The effect of clustering is seen in figure 5.6 where also the coloured visualization of pieces is enabled. Clustering as an idea have been influenced by Houdini [6] which supports a similar feature.

5. Result

The results presented in this chapter was gathered on a test system running Maya 2011 in a Linux environment. The test system was a laptop with an Intel i5-520M 2.4GHz CPU with 4 available threads and 4GB of 1333MHz DDR3 memory.

By using pre-fracturing a multitude of results can be achieved to mimic physical properties. The need for physically correct results is less important than the freedom to model and animate specific pieces in non-correct ways to create the sought for artistic result. When viewing the results it needs to be considered that these are only the initial geometries and further detail is added with textures and shaders upon rendering of the results.

5.1. Random

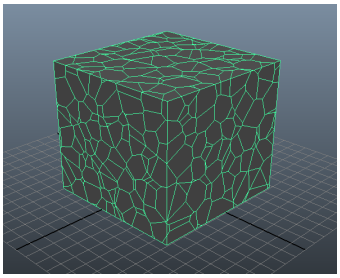
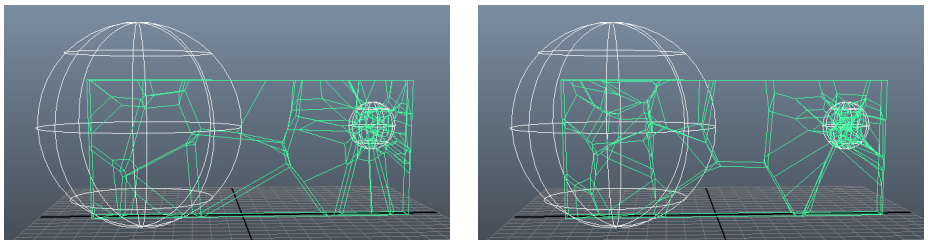


Figure 5.1.: Random sampling. 1000 site points.

Standard random sampling as in figure 5.1 of site points creates an almost uniform look that can be seen as repetitive. All pieces get a very similar look and size. It is however a very quick way to quickly shatter an object into an desired amount of pieces without manually specify areas that should have a low or high density of site points.

5.2. NURBS Sphere

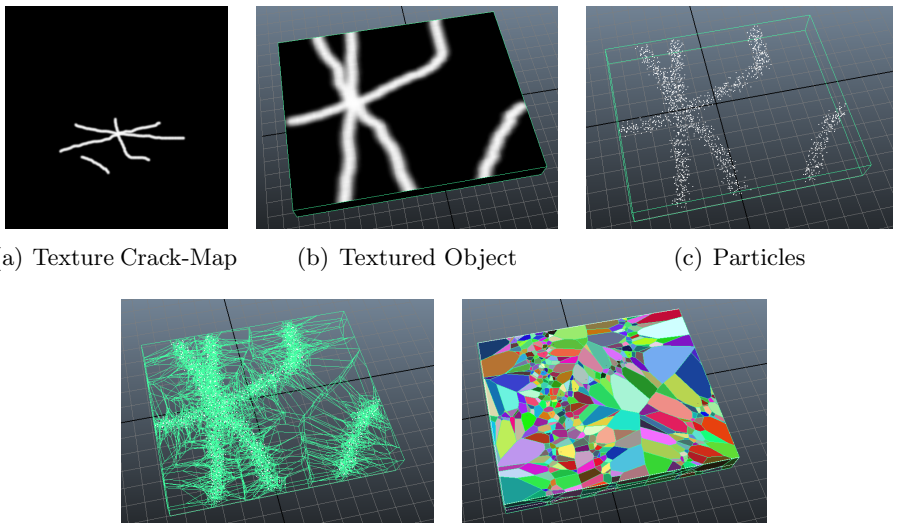


(a) Default NURBS sphere sampling (b) Exponential NURBS sphere sampling

Figure 5.2.: NURBS sphere sampling of points

The possibility of directing the positioning of the site points make it possible to in a simple and fast way making for example the middle of a pillar break into an high amount of small pieces while creating bigger pieces for the rest of the pillar. This can also be achieved by first separating the middle of the pillar but if there are multiple regions of the pillar where this needs to be done the work can become tedious. A NURBS sphere set-up is shown in figure 5.2.

5.3. Particles and Crack-Maps



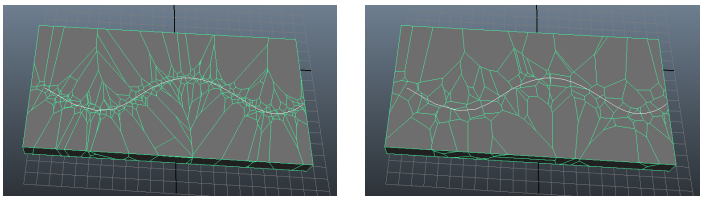
(a) Texture Crack-Map (b) Textured Object (c) Particles

(d) Fractured using particles (e) Resulting pieces with colour visualization

Figure 5.3.: Particles as site points

Through utilizing Maya particles and emission based on a texture crack patterns can be painted on existing UVs where additional info about the object that is not visible in the mesh is stored. Particles can also be emitted from collision points to create impact fracture around a certain area. Figure 5.3(a) of figure 5.3 shows the texture in UV space that is applied to the object. The crack-map in this case only contains information for the top of the object and when applied produces figure 5.3(b). The particles are emitted in negative direction of the surface normal into the object and produces the result in figure 5.3(c). After fracturing the wire-frame 5.3(d) and colour visualized 5.3(e) is achieved containing smaller pieces in the areas of where the original texture specified.

5.4. NURBS Curve

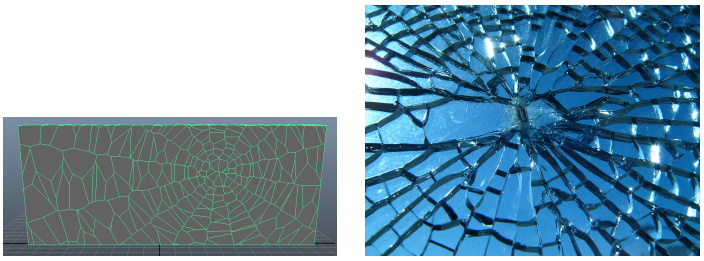


(a) Short step size (b) Long step size

Figure 5.4.: Nurb curve sampling of points

If an effect simulating a ground breaking up it can be created by drawing out some curves on the object where the ground should tear up. When simulating a static object is used to either push the pieces upwards or rules can be set-up to gradually drop the pieces along an axis as time progresses. By modifying the step size to be larger along the curve the width of the crack can be controlled to create wider destruction as in figure 5.4(b) and by setting a smaller step size it can be narrowed to produce figure 5.4(a).

5.5. Radial Shatter

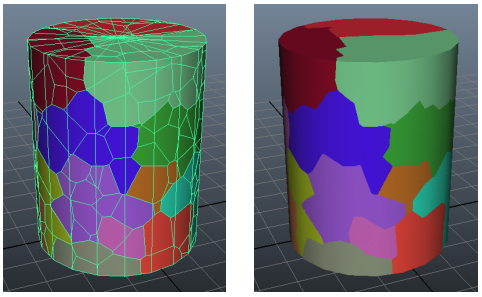


(a) Viewport (b) Photo reference

Figure 5.5.: Radial shatter sampling.

Radial sampling mimics the pattern which is creating when a glass window is fractured. It can also be used for crater like patterns upon impact with a ground object. A comparison between an actual broken window in figure 5.5(b) and radial sampling of Voronoi site points in figure 5.5(a) shows the similarities.

5.6. Clustering and Visualization



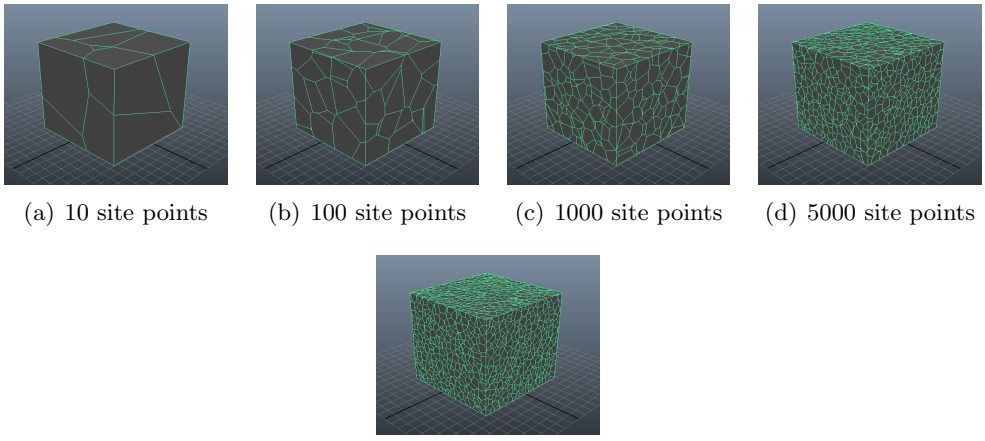
(a) With wireframe (b) Without wireframe

Figure 5.6.: Clustering pieces

Clustering of pieces is a good way of suppressing the classical Voronoi look that can be seen in many cases. It is an extra tool that can be used together with regular displacement shaders to create realistic looking results.

The visualization of pieces in random colour makes them easy to distinguish between each other both in the case of regular fracturing but especially when clustering is used as cuts for the pieces that make up the bigger pieces are also visible on the wire-frame.

5.7. Performance



(a) 10 site points (b) 100 site points (c) 1000 site points (d) 5000 site points

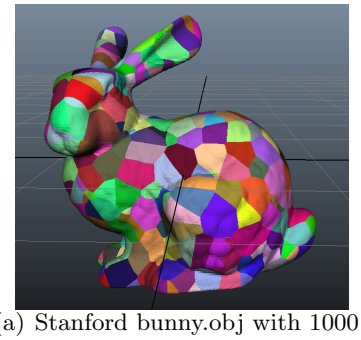
(e) 10000 site points

Figure 5.7.: Random sampling of site points

site points	*voro (sec)*	*dg_voro [5] (sec)*	*Houdini 11.1[6] (sec)*
10	0.03	1.18	0.05
100	0.07	61.00	0.10
1000	2.34	3897.73	1.96
5000	55.04	x	13.52
10000	153.82	x	41.03

Table 5.1.: Speed result comparison. Random sampling.

The performance towards freely available solutions such as [5] is great. If comparing against a commercial solution as Houdini the implementation falls behind, mostly because of the Voronoi implementation of Houdini being threaded and able to utilize all threads of the test-system. Further discussion about threading the implementation is found in section 7.1. Comparing towards commercial solutions for Maya have not been possible because of the prices of these products or that they are not available except for invited clients.



(a) Stanford bunny.obj with 1000 site points.

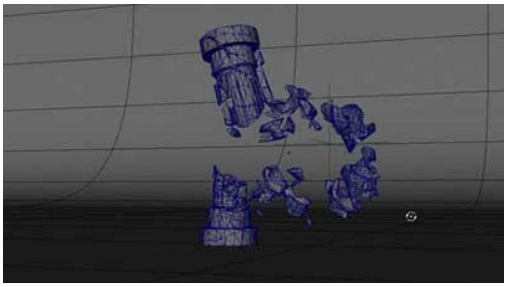
site points	*voro (sec)*
10	3.38
100	31.97
500	298.94
1000	936.37

(b) Speed result high polygon object. Stanford bunny.obj 73173 polygons.

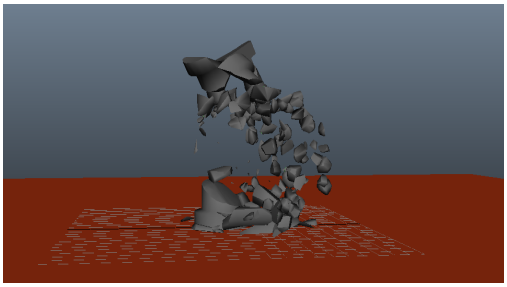
Figure 5.8.: Fracturing comparison

Because of the speed increase compared to other available solutions fairly high polygon objects can be fractured in a reasonable time. Table 5.8(b) contains running times for fracturing the Stanford bunny in figure 5.8(a) into various amounts of pieces.

5.8. Simulation Fracturing Comparison



(a) Simulation based fracturing using DMM [22].

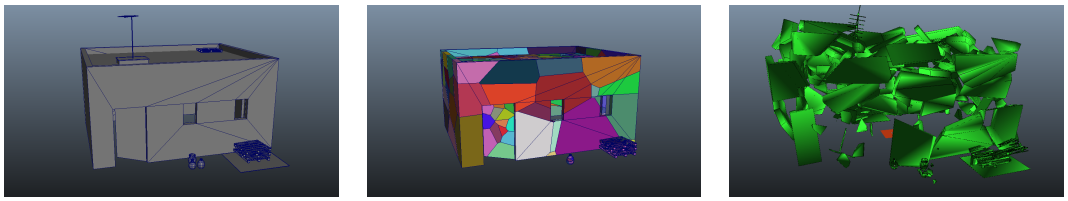


(b) Pre-fracturing using voro

Figure 5.9.: Fracturing comparison

To provide a comparison between the thesis implementation and the techniques discussed in section 2.2 a figure 5.9(a) is taken from an Autodesk Tutorial on Breaking Pillars using DMM [22] that was introduced in Maya 2012. A simple pillar was modelled in Maya and a quick set-up using pre-fracturing technique was set-up to mimic the result. The result can be seen in Figure 5.9(b).

5.9. In Use



(a) Original Scene (b) Fractured Geometry (c) RBD Simulation using Dynamica [2] and Maya Fields

Figure 5.10.: In use example

Figure 5.9 showcases how the processes of loading an initial scene, fracturing it and simulating it can look in the view-port of Maya. These in use examples were created for demonstration purpose and to produce a realistic result more time needs to be put into both fracture modelling and setting up the simulation.

6. Discussion

A drawback of the methods chosen for implementation 3.3 in the thesis is foremost it's problem with handling non-manifold surfaces. This could be solved by implementing a pre-step that makes sure that the object is manifold but most objects to be fractured are usually of simpler and cleaner shapes not only to make the fracturing process easier but to make the Rigid Body Simulations faster. Both if using volume based collisions or polygonal collision the calculation speed is greatly increased the lower polygon count and simpler shape an object consists of.

In the speed comparison to commercial plug-ins and programs in table 5.1 section 5.7 it can be noted that the commercial product Houdini have roughly a 4x speed improvement compared to the implementation of the thesis. By monitoring system usage it can be easily observed that Houdini uses all 4 available threads of the test system while the thesis implementation is limited to one. More on this is found in section 7.1.1. Especially the fracturing of the high polygon objects would greatly benefit from threading of the plug-in.

It can be further discussed how long it will be before speed of simulated fracturing is good enough to be used in large scale destructions and what influence it will have on the currently most common work-flows. Pixelux [1] have interesting technology but there are still no results where the potential have been fully shown and it suffers from a similar pattern as pre-fracturing with Voronoi where the pattern can be distinguished for the trained eye. With technology based on FEA (Finite Element Analysis) the tetrahedrons instead become the pattern to be distinguished if the resolution is not high enough. The comparison in 5.8 also shows that the results of using a simulation based method are very similar to a directed pre-fractured approach and upon addition of dust, debris, motion blur fine details that matters won't be distinguishable.

An important but not easy to measure in results part of the thesis was the work-flow and use of the plug-in. Fracturing multiple objects simultaneously, keeping original surface properties, not cluttering the interface and most importantly not crashing the system even if used on a bad geometry. Worst case scenario the implementation fails to cap the geometry in certain areas but great detail have been put into avoiding numerical errors and endless loops to ensure that it is possible to keep working without losing important content.

7. Conclusion

The plug-in implemented for this thesis is production ready and is a great improvement to currently used and available techniques but can be improved in several ways to make the work-flow simpler and the performance higher.

Because of all important code implemented being separate and not dependent on Maya classes except for reading and creating the meshes and certain information for sampling points the plug-in could be easily updated to support newer versions of the Maya API if the syntax, where to change, and it could also be implemented into other software's.

7.1. Further Work

To further increase speed and usability of the plug-in there are several steps and actions that can be taken as threading, more procedural-ism, more options for cut planes and a Maya command implementation to simplify including the plug-in in scripts.

7.1.1. Threading

Because Maya still being compiled for Linux using a fairly old version of gcc (4.1.2) the number of available solutions for threading is limited to pthread (POSIX Threads). pthread is not as straight forward to implement as newer solutions as OpenMP.

The method used for the fracturing is easy to compute threaded as each piece of the fractured object is generated from the same initial data. If done this would increase speed greatly when generating high amount of pieces.

7.1.2. Procedural fracture

For making the set up process of large scenes of destruction it would be ideal to be able to set objects as breakable in a similar way that is implemented in Houdini [6]. By considering the strength and position of the impulse force upon collision site points are scattered in the object and used to generate new pieces that can further break upon another collision. This procedural way of fracturing limits the possibility to direct the fracturing but is very usable when a lot of objects need to break.

The biggest problem for implementing this is the current implementation of Bullet for Maya (Dynamica [2]) and would need customization to support this. Another approach is to integrate the Bullet solved into the actual plug-in but it would require a bigger amount of work.

7.1.3. Arbitrary Cut Planes

Further detail of modelling could be achieved by supporting arbitrary planes to be used as cut objects instead of only supporting flat planes. Being able to use a plane with noise displacement would help to hide the Voronoi pattern in the pieces but also greatly increase computation time.

7.1.4. Command Implementation

To further extend usability implementing a complementary command instead of node version for Maya would be a useful extension. This would enable artists to execute the algorithm in scripts and other ways that are currently not possible. For example objects could be selected and a script could be ran that fractured each object into random amount of objects within a specified range and producing ready to simulate pieces without any additional set-up.

Acknowledgements

I would like to thank Everyone at ILP for letting me do the thesis with them. Especially Eric Hermelin for helping with wrapping my head around the Maya API and Yafei Wu for important work-flow suggestions and improvements.

Bibliography

[1] DMM by Pixelux. http://www.pixelux.com/

[2] Dynamica. http://code.google.com/p/dynamica/

[3] Autodesk Maya API. http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=9469002

[4] OpenGL Mathematics http://glm.g-truc.net

[5] dg_Voro_Py 1.0.0 *mayascript*. http://www.creativecrash.com/maya/downloads/scripts-plugins/dynamics/c/dg_voro_py--2

[6] Side Effect Software Inc. Houdini 11.1 http://www.sidefx.com/.

[7] Eric Haines. *Point in Polygon Strategies*. Graphics Gems IV, ed. Paul Heckbert, Academic Press, p. 24-46. 1994. http://erich.realtimerendering.com/ptinpoly/

[8] David Eberly. *Clipping a Mesh Against a Plane*. Geometric Tools, LLC, http://www.geometrictools.com/. 2002,2008.

[9] Rachel Weinstein, Frank Petterson, Brice Criswell. *Destruction System*. ACM SIGGRAPH 2008 Sketch. 2008.

[10] Kevin Weiler, Peter Atherton. *Hidden Surface Removal Using Polygon Area Sorting*. Program of Computer Graphics, Cornell University, Ithaca, New York 14853. 1977.

[11] Donald B. Johnson *Finding All The Elementary Circuits Of A Directed Graph*. SIAM J. COMPUT. Vol. 4, No. 1, March 1975.

[12] Saty Raghavachary *Fracture generation on polygonal meshes using Voronoi polygons* ACM SIGGRAPH 2002 Sketch. 2002.

[13] Paeth, A.W. *Graphics gems V* 9780125434553, The graphics gems series, AP Professional. 1995.

[14] David Kirk. *Graphics gems III* 978-0124096738, Morgan Kaufmann. 1994.

[15] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, Cambridge, 1990.

[16] Georgy Voronoi. *Nouvelles applications des paramètres continus à la théorie des formes quadratiques*. Journal für die Reine und Angewandte Mathematik, 133:97-178, 1908.

[17] Eric Haines. *Essential ray tracing algorithms* from *An Introduction to Ray Tracing*. Academic Press, New York. 1989.

[18] J. C. TIERNAN. *An efficient search algorithm to find the elementary circuits of a graph*. Comm. ACM, 13 (1970), pp. 722-726.

[19] Nafees Bin Zafar, Mårten Larsson, Ryo Sakaguchi, Brian Gazdik *Procedural Methods For Large Scale Destruction* ACM SIGGRAPH 2011 Talk. 2011.

[20] JF O'Brien, J.K. Hodgins. *Graphical Modeling and Animation of Brittle Fracture*. SIGGRAPH 99 Conference Proceedings. 1999.

[21] JF O'Brien, J.K. Hodgins. W.A. Bargteil. *Graphical Modeling and Animation of Ductile Fracture*. ACM Trans. Graph. 2002.

[22] Breaking Pillars with DMM in Maya 2012. `http://area.autodesk.com/tutorials/breaking_pillars`

[23] Ivan Sutherland, Gary W. Hodgman. *Reentrant Polygon Clipping*. Communications of the ACM, vol. 17, pp. 32-42, 1974

A. User Manual

A.1. Parameters

Parameter	*Explanation*
Seed Value	Seed value for random number generation
Random Points	Amount of random site points
Curve Steps	Amount of steps to take along the NURBS-curve
Curve Points	Amount of random points at each step
Sphere Points	Amount of points to generate inside NURBS-spheres
Use exponential sphere points	Enable or disable exponential sampling
Increment seed value	Increment seed value for each time Update is pressed
Use time as seed	Use current time as seed to always produce unique result
Use original normals	Transfer the normals from the original object to the generated pieces
Shell only	Do not create inner surfaces using capping
Clustering	Enable or disable clustering
Cluster Points	Amount of clusters to be generated
Radial Axis	Axis along which the radial points will be sampled
Radial Iter A	Amount of rings for the pattern
Radial Iter B	Amount of cuts in the rings for the pattern
Radial Exp	Exponentially grow step size between rings
Radial Random	Amount of random jitter to add to points
Radial Step Size	Step size between each ring
Radial Start Step Size	Step size from centre point to first ring
Offset	Offset the enter point by these coordinates

Table A.1.: User interface parameters explained

A.2. Usage

For all use cases the initial step is to run the voro command on selected objects. If the object is in Maya considered mesh object it will create the voroNode and set-up necessary

connections. When the desired look have been achieved pieces are baked to separate geometry using the voroDone command or by using Maya internal *Mesh Separate* option. voroDone is faster and needs to be used on objects containing a high amount of pieces.

A.3. Random and Radial Mode

For the usage of Random sampling and Radial sampling it is merely to configure the settings of the voroNode and updating until satisfied.

A.4. NURBS Sphere, NURBS Curve and Particle Mode.

NURBS sphere, NURBS curve and Particle mode are activated by view-port selection. By selecting either a sphere, curve or particle emitter together with the objects currently being fractured the voroNode reads the information from the objects and calculates the site points.